Triangle meshes

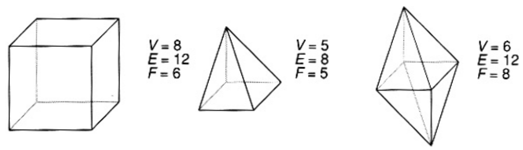
CS 465 Lecture 7

Acknowledgement

- Most slides: Steve Marschner

Notation

- n_T = #tris; n_V = #verts; n_E = #edges
- Euler: $n_V - n_E + n_T = 2$ for a simple closed surface
  - and in general sums to small integer
  - argument has implication that $n_T:n_E:n_V$ is about 2:3:1



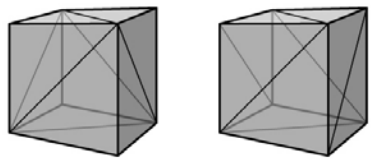
[Foley et al.]

Validity of triangle meshes

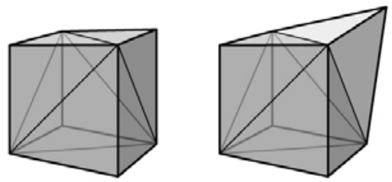
- in many cases we care about the mesh being able to bound a region of space nicely
- in other cases we want triangle meshes to fulfill assumptions of algorithms that will operate on them (and may fail on malformed input)
- two completely separate issues:
  - topology: how the triangles are connected (ignoring the positions entirely)
  - geometry: where the triangles are in 3D space, i.e., the embedding of the mesh in 3-space.

Topology/geometry examples

- same geometry, different mesh topology:

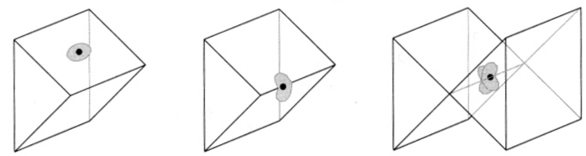


- same mesh topology, different geometry:



Topological validity

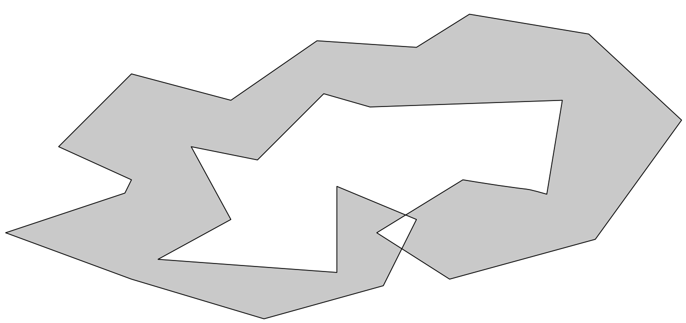
- strongest property, and most simple: be a **manifold**
  - this means that no points should be "special"
    - Neighborhood of each point is topologically equivalent to a disk.
  - interior points are fine
  - edge points: each edge should have exactly 2 triangles
  - vertex points: each vertex should have one loop of triangles
    - not too hard to weaken this to allow boundaries



[Foley et al.]

Geometric validity

- usually want non-self-intersecting surface
- hard to guarantee in general
  - because far-apart parts of mesh might intersect



Representation of triangle meshes

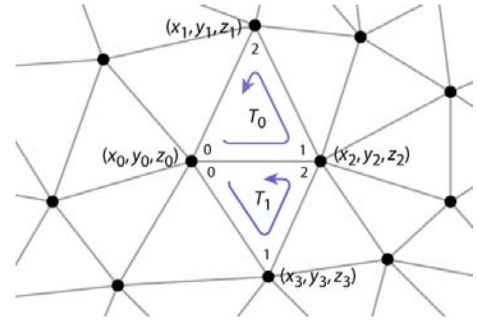
- Compactness
- Efficiency for rendering
  - enumerate all triangles as triples of 3D points
- Efficiency of queries
  - all vertices of a triangle
  - all triangles around a vertex
  - neighboring triangles of a triangle
  - (need depends on application)
    - finding triangle strips
    - computing subdivision surfaces
    - mesh editing and cutting
    - physical simulation

Representations for triangle meshes

- Separate triangles
- Indexed triangle set
  - shared vertices
- Triangle strips and triangle fans
  - compression schemes for transmission to hardware
- Triangle-neighbor data structure
  - supports adjacency queries
- Winged-edge data structure
  - supports general polygon meshes

Separate triangles

	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮

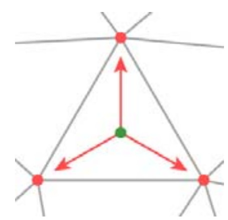


Separate triangles

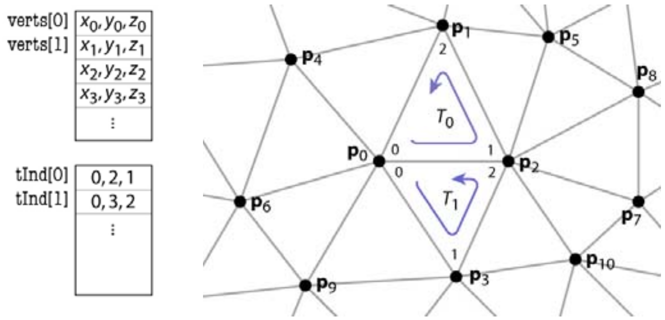
- array of triples of points
  - float[n_T][3][3]: about 72 bytes per vertex
    - 2 triangles per vertex (on average)
    - 3 vertices per triangle
    - 3 coordinates per vertex
    - 4 bytes per coordinate (float)
- various problems
  - wastes space (each vertex stored 6 times)
  - cracks due to round-off
  - difficulty of finding neighbors at all

Indexed triangle set

- Store each vertex once
- Each triangle points to its three vertices



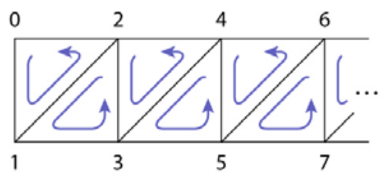
Indexed triangle set



Indexed triangle set

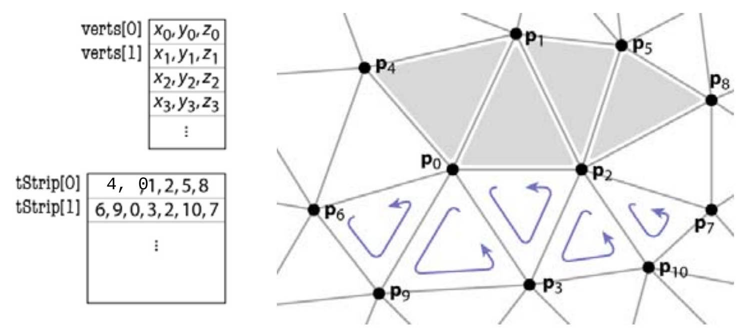
- array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
    - (3 coordinates x 4 bytes) per vertex
- array of triples of indices (per triangle)
  - int[n_T][3]: about 24 bytes per vertex
    - 2 triangles per vertex (on average)
    - (3 indices x 4 bytes) per triangle
- total storage: 36 bytes per vertex (factor of 2 savings)
- represents topology and geometry separately
- finding neighbors is at least well defined

Triangle strips



- Take advantage of the mesh property
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by reusing the second and third vertices of the previous triangle
  - every sequence of three vertices produces a triangle (but not in the same order)
  - e.g., 0, 1, 2, 3, 4, 5, 6, 7, … leads to (0 1 2), (2 1 3), (2 3 4), (4 3 5), (4 5 6), (6 5 7), …
  - for long strips, this requires about one index per triangle

Triangle strips

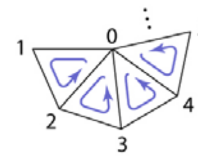


Triangle strips

- array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
    - (3 coordinates x 4 bytes) per vertex
- array of index lists
  - int[n_S][*variable*]: 2 + *n* indices per strip
  - on average, (1 + ε) indices per triangle (assuming long strips)
    - 2 triangles per vertex (on average)
    - about 4 bytes per triangle (on average)
- total is 20 bytes per vertex (limiting best case)
  - factor of 3.6 over separate triangles; 1.8 over indexed mesh

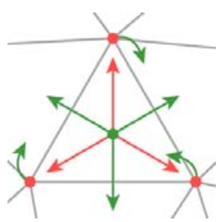
Triangle fans

- Same idea as triangle strips, but keep oldest rather than newest
  - every sequence of three vertices produces a triangle
  - e. g., 0, 1, 2, 3, 4, 5, … leads to (0 1 2), (0 2 3), (0 3 4), (0
  - for long fans, this requires about one index per triangle
- Memory considerations exactly the same as triangle strip

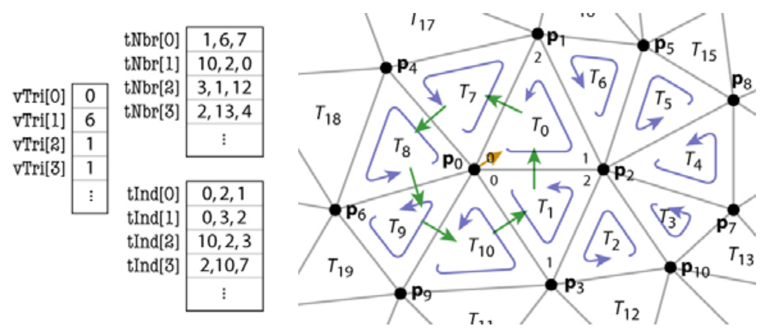


Triangle neighbor structure

- Extension to indexed triangle set
- Triangle points to its three neighboring triangles
- Vertex points to a single neighboring triangle
- Can now enumerate triangles around a vertex



Triangle neighbor structure

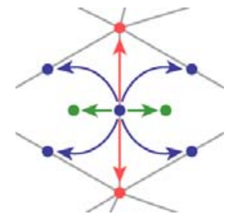


Triangle neighbor structure

- indexed mesh was 36 bytes per vertex
- add an array of triples of indices (per triangle)
  - int[n_T][3]: about 24 bytes per vertex
    - 2 triangles per vertex (on average)
    - (3 indices x 4 bytes) per triangle
- total storage: 60 bytes per vertex
  - still not as much as separate triangles

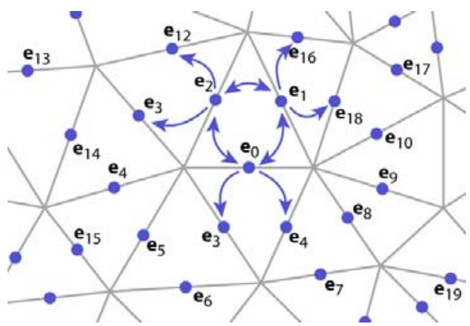
Winged-edge mesh

- Edge-centric rather than face-centric
  - therefore also works for polygon meshes
- Each (oriented) edge points to:
  - left and right forward edges
  - left and right backward edges
  - front and back vertices
  - left and right faces
- Each face or vertex points to one edge



Winged-edge structure

	hl	hr	tl	tr
edge[0]	1	4	2	3
edge[1]	18	0	16	2
edge[2]	12	1	3	0
		⋮		

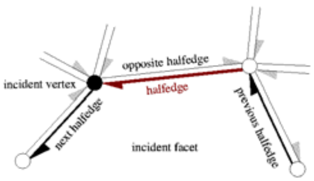


Winged-edge structure

- array of vertex positions: 12 bytes/vert
- array of 8-tuples of indices (per edge)
  - head/tail left/right edges + head/tail verts + left/right tris
  - int[n_E][8]: about 96 bytes per vertex
    - 3 edges per vertex (on average)
    - (8 indices x 4 bytes) per edge
- total storage: 108 bytes per vertex
  - so this is more complex than neighbor pointers
  - but it is cleaner and generalizes to polygon meshes

Half-edge Mesh

- Edge-centric rather than face-centric
- Half-edge version of winged-edge
- Each (oriented) half-edge points to:
  - next edge
  - opposite edge
  - head vertex
  - left face



[From CGAL]

- Each face points to one edge
- Each vertex points to one incident edge
- Same storage requirements as winged-edge
  - Different interface

Mesh Compression

- **Increasingly desirable**
- **Getting better all the time:**
  - **Topology (<1 bit /vertex)**
  - **Geometry (<6 bits/vertex)**

St. Matthew scan

(>370 million triangles)

The Digital Michelangelo Project